Predicting and Tuning the Performance of Multicore Systems

Donald E. Porter, Owen S. Hofmann, and Emmett Witchel
Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712
{porterde, osh, witchel}@cs.utexas.edu
February 29, 2007

ABSTRACT

This paper introduces a novel method and a tool called *Syncchar* for understanding and modeling the performance of shared memory parallel programs. Syncchar helps programmers choose the type of synchronization they need and focuses their attention on which critical regions would benefit most from reorganization. Conservative synchronization (e.g., a spinlock) performs best for highly contended data structures, whereas optimistic synchronization (e.g., transactional memory or a sequence lock) performs best for lightly contended data structures.

Syncchar uses *data independence* and *conflict density* to model the performance of parallel programs. A pair of critical regions are data independent if the memory written by one critical region (its write set) does not intersect the memory read or written by the other critical region (its address set). When critical regions are not data independent, conflict density measures the degree to which there is a single "problem" thread that is conflicting with many other threads, or whether all conflicting threads conflict with each other. By sampling the address sets of a lock-based program, the Syncchar algorithm predicts the speedup of the program if it were converted to use optimistic concurrency. Syncchar also identifies contention hot-spots in critical sections, which helps focus programmer effort in the areas of greatest performance gains.

The paper measures and validates the Syncchar model using several microbenchmarks, and provides a case study of performance tuning critical regions in the Linux kernel. The Syncchar tool uncovers opportunities to increase the performance of the Linux kernel by as much as 4.8% and the transactional Linux kernel (TxLinux) by up to 8%.

1. INTRODUCTION

Multicore architectures are now the norm, and most processor manufacturers plan to scale the number of cores on a die with successive processor generations. The end-user benefit of these systems will be limited by how well average software developers can effectively leverage the parallel hardware provided by these new processors. Scaling the concurrent performance of irregular end-user applications remains difficult despite nearly fifty years of research and product development. Determining the optimal size and structure of critical sections remains a black art.

Concurrent programming in a shared memory system requires primitives such as locks to synchronize threads of execution. Locks have many known problems, including deadlock, convoying, and priority inversion that make concurrent programming in a shared memory model difficult. In addition, locks are a *conservative* synchronization primitive—they always assure mutual exclusion, regardless of whether threads actually need to execute a critical section sequentially for correctness. Consider modifying elements in a binary search tree. If the tree is large and the modifications are evenly distributed, most modifications can safely occur in parallel. A lock protecting the entire tree will needlessly serialize modifications.

One solution for the problem of conservative locking is to synchronize data accesses at a finer granularity–rather than lock an entire binary search tree, lock only the individual nodes being modified. This presents two problems. First, data structure invariants enforce a lower bound on the locking granularity. In some data structures, this bound may be too high to fully realize the available data parallelism. Second, breaking coarse-grained locks into many fine-grained locks significantly increases code complexity. As the locking scheme becomes more complicated, long-term correctness and maintainability are jeopardized.

An alternative to conservative locking is *optimistic concurrency*. An optimistic system allows accesses to shared data to proceed concurrently, dynamically detecting and recovering from conflicting accesses. Transactional memory provides hardware or software support for designating arbitrary regions of code to appear to execute with atomicity, isolation and consistency [9, 11]. Transactions provide a generic mechanism for optimistic concurrency by allowing critical sections to execute concurrently and automatically rolling back their effects on a data conflict. Coarse-grained transactions are able to reduce code complexity while retaining the concurrency of fine-grained locks.

The problem for the average programmer is the difficulty of knowing what kind of concurrency control to use. A lock-based program might not see any speedup upon conversion to use transactions—a fact software engineers and managers would like to know in advance. Understanding concurrent performance is difficult and tools are needed to make good decisions for software development. An application's parallel speedup might be just around the corner—if you look around the right corner.

This paper presents *Syncchar*, a tool and methodology for understanding and performance tuning concurrent programs. Syncchar samples the sets of addresses read and written while locks are held. It then builds a model of the program's execution that can predict the performance of the application if it were converted to

TR-08-10 Department of Computer Sciences, The University of Texas at Austin

use optimistic synchronization. The model has two parts, the *data independence* and *conflict density* of the critical regions. Data independence measures how likely it is that threads will access disjoint data, while conflict density measures how many threads are likely to be involved in a data conflict should one occur.

Concurrent performance tuning is complicated by the fact that the intuitions and techniques that programmers have developed in tuning lock-based code will not be sufficient to identify and correct data access patterns that throttle the performance of optimistic systems. For instance, common lock-based programming techniques, such as walking a linked list or incrementing a shared counter inside a critical region, can negatively impact optimistic performance. Syncchar focuses programmer attention on code and data structures where reorganization will increase concurrent performance.

This paper presents background on optimistic concurrency (Section 2) and describes the need for a performance prediction and tuning tool (Section 3). The paper then presents novel techniques and a tool for investigating the potential of optimistic concurrency by measuring the data independence and conflict density of critical regions that are protected by the same lock (Section 4). The paper then validates the model with a set of microbenchmarks (Section 5), and applies the methodology and tool to tune the performance of the TxLinux kernel [26, 27] (Section 6). Section 7 presents related work, and Section 8 concludes.

2. BACKGROUND ON OPTIMISTIC CONCURRENCY

This section presents definitions and terminology for discussing conservative and optimistic concurrency.

Optimistic concurrency control (e.g., transactional memory) often relies on *conflict serializability* as its safety condition. We call the set of addresses read during a critical section, the *read set*, the set of addresses written the *write set* and the union of read and write set the *address set*. For critical sections A and B, A conflicts with B if:

$$A_W \cap (B_R \cup B_W) \neq \emptyset$$

Informally, conflict serializability says that the write set of one critical region must be disjoint from the other's address set to guarantee safety. Conflict serializability is efficient to compute so it is used widely in transactional memory systems.

Conflict serializability is a pessimistic model because two critical sections can conflict, yet safely execute concurrently. For example, if two critical sections conflict only on their final write, they can still safely execute concurrently if one finishes and commits before the other issues the conflicting write. Some data structures and algorithms make stronger guarantees that allow critical sections to write concurrently to the same locations safely, but these are relatively rare and beyond the scope of this paper.

As an example of conflict serializability, consider the simple binary tree in Figure 1. Three different critical sections that operate on this tree, along with their address sets are listed in Table 1. Critical sections 1 and 2 read many of the same memory locations, but only write to locations that are not in the other's address sets. They are, therefore, *data independent* and could safely execute concurrently. This makes sense intuitively because they modify different branches of the tree. Critical section 3, however, modifies the left pointer in the root of the tree, which cannot execute concurrently with critical sections that operate on the left branch of the tree. This is reflected in the address sets: 0x1032 is in Critical Section 3's write set and in the read sets of Critical Sections 1 and 2.

Critical regions are data independent if their write set is disjoint from the other's address set. If critical sections concurrently modify the same data, or have *data conflicts*[1], optimistic concurrency control will serialize access to critical sections. In such cases optimistic control can perform much worse than conservative locking due to the overhead required to detect and resolve conflicts.

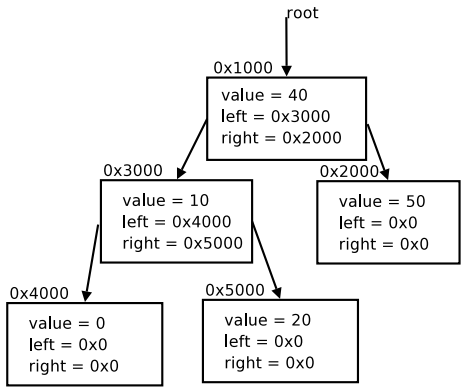

Figure 1: A simple binary tree.

In our simple example, we can determine the data independence of the critical sections by inspection. In most cases, however, this is insufficient because functions that modify a data structure generally determine what to modify based on an input parameter. Thus, the data independence of critical section executions largely depends on the program's input, requiring investigation of the common case synchronization behavior in order to determine whether to employ optimistic concurrency.

In converting lock based code to use optimistic concurrency, one must be aware that many common idioms in lock-based programming can needlessly limit concurrency under optimistic synchronization. For example, if two critical regions operate on completely separate data except for updating some shared performance statistics at the end of the critical section, these critical regions will have to serialize on their updates shared statistics fields. These conflicts are not related to the data structure, and can be avoided by removing or restructuring the statistics data.

Conflict density is a measure of the connectedness of the graph for a data conflict. Assume a conflict among N threads. In the best case, a single thread might write a datum read by $N-1$ other threads. This is a low density conflict that produces a short serialized execution schedule ($N-1$ readers commit, and then the writer). In the worst case, each thread can write a datum written by every other of the $N-1$ threads, yielding a high density conflict that necessitates a completely sequential schedule (the threads must run serially, one after the next).

The discussion of optimistic concurrency in this paper primarily focuses on transactional memory, as transactions are the most general purpose optimistic programming model. There are, however, other forms of optimistic concurrency, discussed in Section 7.2.

3. TUNING CONCURRENT PERFORMANCE IS DIFFICULT

This section motivates a tool for understanding the performance of concurrent systems by discussing the difficulty of performance tuning concurrent programs.

Although a conversion from locks to transactional memory can be straightforward, there are a number of pitfalls that can require

[1]We selected the term data conflicts over data dependence to avoid confusion with other meanings.

Critical Section 1			Critical Section 2			Critical Section 3	
begin critical section; node = root→right; node→left = root→left→right; end critical section;			begin critical section; node = root→left; node→left = root→left→right; end critical section;			begin critical section; node = root; node→left = node→right; end critical section;	
r		w	r		w	r	w
0x1000	0x2064	0x3032	0x1000	0x2064	0x2032	0x1000	0x1032*
0x1032*	0x3000		0x1032*	0x3000		0x1064	
0x1064	0x3064		0x1064	0x3064			
0x2000			0x2000				

Table 1: Three critical sections that could execute on the tree in Figure 1 and their address sets. The read entries marked with an asterisk (*) are conflicting with the write in Critical Section 3.

substantial engineering effort to resolve. Predicting the benefits of such a conversion before a single line of code is modified will allow engineers and product managers to make a more informed decision about adopting a transactional memory system. Further, if performance of a converted system does not meet expectations, a predicted speedup helps assess whether the problem lies with the application or the transactional memory implementation. The transactional memory programming community can also benefit from a predictor of performance in developing standard benchmarks for transactional memory implementations and reasoning about the performance of a particular system.

Tuning the performance of lock-based programs generally involves either identifying highly contended locks and breaking them down into smaller locks, or restructuring the data to avoid synchronization (e.g., per-thread data structures and read-copy update). Tuning optimistic synchronization, on the other hand, requires a reduction in conflicting memory operations. Although there are some known techniques for avoiding memory conflicts, it can be a difficult and non-obvious process. The following subsections illustrate three key challenges that require Syncchar's quantitatively driven approach to performance tuning is needed for the programmer to most efficiently use her time.

3.1 Data sharing hot-spots

A common optimization in lock-based programming is caching attributes such as the number of elements in a data structure. Since the lock protecting the data structure is already held, there is no synchronization overhead to updating these integer fields—with the added benefit of saving work when this information is needed later. In an optimistic model, however, concurrent execution must be serialized on these fields. This can be particularly prone to conflict and wasted work if updates are made late in the critical section and values are read early in others.

One solution for avoiding conflicts on counters that cannot be eliminated is splitting them into per-thread counters. This solution avoids conflicts on writes, but introduces conflicts on reads, as each CPU's value must be read to provide a correct sum. Per-CPU counters thus work well if they are updated more frequently than they are read, but are pointless otherwise.

Linked lists are commonly used as a generic container because they are simple to implement, have low memory overhead, and don't have problems with resizing. Operating systems use linked lists extensively. For optimistic synchronization, however, they can be pathologically bad, as each thread traverses the exact same path of pointers inside of a critical region. Any pointer update will conflict with all other critical regions that have walked further down the list, despite the fact that concurrent execution of the operations should be semantically safe.

The Syncchar tool can identify data sharing hot-spots that will limit optimistic concurrency, focusing programmer effort on where the potential returns are the greatest.

3.2 Logical isolation vs. physical isolation

A common solution for avoiding low-level conflicts in the database literature is *open nesting* [19]. When an open nested transaction commits, the transactional memory system no longer retains physical isolation on the nested transaction. The onus of providing logical isolation at a higher level is shifted to the programmer. By releasing physical isolation on low-level, conflict prone operations, the restart rate can be substantially lowered in some cases. Open nested transactions may require the programmer to write special handlers for commit and abort of the parent transaction to release the logical isolation or roll back the open nested transaction, respectively. Open nesting implementations have been proposed for hardware transactional memory systems [18]. There are also hardware proposals for transactional suspension mechanisms that suspend physical isolation altogether, requiring that operations be synchronized through other means such as locks [31]. The Galois system [10] and Transactional Boosting [7] provide more structured variants of open nesting in software that require sophisticated reasoning by the data structure implementer about properties of the operations such as associativity and linearizability. These variants are easier for other programmers to incorporate into a transaction than writing open nested transactions.

The primary disadvantage to techniques such as open nesting and transactional suspension is that it is difficult for programmers to reason about the correctness of the logical isolation mechanisms and commit and abort handlers. For this reason, these techniques should only be attempted when there is a strong indication that they will substantially improve performance. By identifying data hot-spots in low-level data structures, Syncchar helps the programmer estimate the benefits of releasing physical isolation.

3.3 Fighting the compiler

A final challenge to optimizing code for optimistic concurrency is compiler optimizations for deeply pipelined machines. Figure 2 illustrates a simple conditional statement the programmer would expect to reduce conflicting accesses to a shared variable. Yet if one inspects the code generated by gcc, it always reads the value from memory and always writes it back. It only uses the condition to determine whether to update the register before writing it back. The compiler is trying to avoid branching around the load and store, which makes sense on a superscalar platform. On a hardware transactional memory system, however, the performance lost to a coherence conflict is much larger than that lost to a mispredicted branch.

```
if(a < threshold){
   shared_variable = new_value;
}
```

```
mov    0x8(%ebp),%edx
cmp    0xc03e6008,%edx
mov    %edx,%eax
cmovge 0xc03e600c,%eax
mov    %eax,0xc03e600c
```

Figure 2: A simple code sequence and the x86 assembly produced by gcc.

For a critical section such as the one above, Syncchar would identify it as having no data independence and flag the shared variable as a contention hot-spot. When the programmer sees results like this that are contrary to his intuition, they serve as a hint to inspect the code generated by the compiler.

Despite a seemingly straightforward appearance, tuning the memory access patterns of an optimistically synchronized system can be subtle and nonintuitive. Even simple rules like making per-thread counters can be pointless or counterproductive when applied by rote. In a large, complicated system, programmers will need tools like Syncchar to identify where their tuning effort is best spent.

4. THE SYNCCHAR MODEL

This section explains the intuitions behind the Syncchar algorithm, and then provides a rigorous treatment of how the algorithm predicts the performance of an optimistically synchronized program based on address sets of critical regions from a conservatively synchronized program.

4.1 Syncchar approach

Assessing the likelihood that critical regions will conflict is at the heart of the Syncchar approach. As discussed above, the performance improvement of a move from lock-based code to optimistic synchronization hinges on the number of critical sections that can successfully execute concurrently.

On one end of the spectrum, data independent critical regions can always execute concurrently, yielding a perfect linear speedup relative to a conservative lock (assuming the regions are equal length). Applications with this characteristic are usually regular enough to attain linear speedup through means other than optimistic concurrency. Thus, the target domain for optimistic concurrency is generally irregular applications that have some measure of data independence. The Syncchar framework provides a way to quantify and measure the data independence of such applications.

Syncchar estimates the data independence and conflict density of critical regions by sampling their address sets. Syncchar collects sample sets of critical regions that could potentially execute concurrently in an optimistic model and determines which of them can conflict and which cannot. This process, described in detail below, is effectively a direct application of the definition of conflict serializability.

By only comparing lock-based critical sections that execute within a limited time window, Syncchar ensures insensitivity to scheduling details and avoids comparisons between samples from entirely different phases of the program's execution. The temporal proximity of critical regions in the lock-based program provides Syncchar with a hint for the temporal proximity of critical regions in an optimistically synchronized version of the program. Because critical section executions that were previously sequential can overlap under optimistic concurrency, the thread schedules are very likely to be different from a lock-based version.

Syncchar only compares critical regions protected by the same lock, across different threads. Critical regions protected by different locks can already execute concurrently in lock-based programs, so they are not the target of the greater concurrency enabled by optimistic synchronization. Because Syncchar does not model thread-level speculation [28], we assume a single thread cannot execute its critical regions in parallel. A single thread might be scheduled on different processors over its lifetime, but this does not affect Syncchar's results.

To avoid spurious conflicts, Syncchar filters a few types of memory accesses from the address sets of critical sections. First, it filters out addresses of lock variables, which are by necessity modified in every critical section. It filters all lock addresses, not just the current lock address, because multiple locks can be held simultaneously and because optimistic synchronization eliminates reads and writes of lock variables. Syncchar also filters stack addresses to avoid conflicts due to reuse of the same stack address in different activation frames.

Syncchar needs to know about dynamically allocated lock variables. If a lock is dynamically allocated, it is a new lock. Some kernel objects, like directory cache entries, are recycled through a cache. The locks in these objects reside at the same address every time they come off the free list, but it is a new lock from a concurrency perspective (critical regions protected by different incarnations of the lock can already execute in parallel). For example, when a directory cache entry emerges from the free pool, its lock is reinitialized and Syncchar considers it a new, active lock. The lock is considered inactive when the object is released back to the free pool. If the lock is reactivated, its address set history is cleared, even though it resides at the same address as in its previous incarnation.

Finally, some spinlocks in the Linux kernel protect against concurrent attempts to execute I/O operations, but do not actually conflict on non-I/O memory addresses. When run on the Linux kernel, Syncchar detects I/O operations, and marks those critical regions as performing I/O. This annotation ensures that these critical regions will not be incorrectly reported as non-conflicting.

4.2 Scheduling model

To predict the speedup of using optimistic concurrency, Syncchar must estimate what critical regions can execute concurrently. Syncchar collects samples of the address sets of each lock. For a given target number of CPUs, Syncchar randomly samples critical regions that executed close to each other in time during the lock-based execution. This method is described below, and pseudocode is presented in Figure 3.

Syncchar tracks all loads and stores executed while a lock is held. When the lock is released, it chooses a random number between zero and one hundred. If the random number is below a certain threshold it records the address set into a buffer, discarding it otherwise. In our experiments we used a sampling rate of 10%. When we compared the results of this to a more exhaustive method, the results varied by less than 1%.

Syncchar also tracks the number of thread identifiers it has collected samples from. Once Syncchar either has samples from its target number of threads, or a full buffer, it performs the data independence and conflict density calculations, described below.

This approach respects the schedule of the original execution, but generalizes it significantly. Determining the exact size of the buffer is somewhat heuristic based on the length of the critical sec-

```
def process_unlock(addressSet, buffer, pids) :
  '''Called after each lock is released'''
  if randint(0,100) < SAMPLE_RATE :
     buffer.append(addressSet)
     if addressSet.pid not in pids:
        pids.append(addressSet.pid)
     if len(buffer) >= BUFFER_MAX \
          or len(pids) == MAX_CPUS :
        compare_addressSets(buffer)
        buffer = []
        pids = []
```

Figure 3: Pseudocode for the Syncchar sampling algorithm.

tions and the scheduling quantum length—one wants a buffer small enough to mostly sample reasonable choices for concurrent critical sections, yet large enough to include critical sections from enough threads to get a good sample. In our microbenchmark experiments, we use a buffer size of 512 address sets.

One advantage of the limited address set buffer is that it limits the complexity of the concurrency model that Syncchar needs. For instance, a newly forked thread cannot execute concurrently with critical region executions that occurred before its creation. Syncchar's sliding window approximates a timeout, restricting which critical sections can be compared. This allows Syncchar to ignore forks and joins, substantially simplifying the scheduling model. By comparing sample critical section executions over a sliding window, Syncchar captures many executions that are likely to be concurrent, while minimizing exposure to error from synchronization through other mechanisms.

4.3 Data independence

Data independence of a lock, I_n, is formally defined as the mean number of threads that will not conflict when n threads are concurrently executing critical sections protected by the same lock.

Note that data independence is a function of the number of threads that can execute concurrently, and not a simple mean. Intuitively, one expects the probability of conflict when only a single thread is scheduled to be zero (data independence of 1), and the likelihood of conflict to increase as the number of threads grows (unless all threads access completely disjoint data). Further, when one has enough CPUs that all possible threads are scheduled, adding more CPUs will not speed up the application any further.

Because data independence varies with the number of CPUs, each data independence calculation must be performed for a target number of CPUs, which we call n. Returning to the sampling methodology described above, Syncchar randomly samples address sets until it obtains n sets from different threads (or it fills its buffer if the maximum available is less than n). This group of address sets is called the sample from the current window. Syncchar assumes that at most n threads will be executing critical regions concurrently. Syncchar does not select more than one critical section per thread for the sample, because it assumes no thread-level speculation occurs.

Syncchar then compares each address set in the sample to all other address sets in the sample, determining which are involved in a conflict C_n and which are not. It keeps a running mean of the number of data independent threads, $I_n = n - |C_n|$. Psuedocode is given in Figure 4.

In the special case that all threads are conflicting, we define data independence to be one rather than zero. Under any reasonable contention management scheme, at least one critical section

```
def compare_addressSets(buffer) :
  sample = choose_n_pids_from(buffer)

  independent = 0
  conflicting = 0
  density = 0
  conflictingAddressSets = []

  # Calculate data independence of sample
  for a in sample :
    for b in (sample - a) :
      if a.compare(b) :
         conflicting += 1
         conflictingAddressSets.append(b)
         break
     if for loop didn't break:
       independent += 1

   # Special case all conflicting threads
   if independent == 0 : independent = 1

   # Calculate conflict density of sample
   for x in conflictingAddressSets :
     local_conflicts = 0.0
     for y in (conflictingAddressSets - x):
       if x.compare(y) :
         local_conflicts += 1
     local_conflicts /= conflicting - 1
     density += local_conflicts

   updateAvg(dataIndependence[n], independent)
   updateAvg(conflictDensity[n], density)
```

Figure 4: Pseudocode for the Syncchar data independence and conflict density calculation.

should be able to complete even in the worst case. By insuring a minimum speedup of one, projections for high-contention workloads under optimistic concurrency approach lock-based performance rather than infinity.

An interesting property of data independence is that one can use it to calculate the probability a thread will be involved in a conflict on the same number of CPUs, $p_{c,n}$. If we treat the number of non-conflicting critical sections as a binomial random variable with parameters $(n, 1 - p_{c,n})$, and the data independence as the expected value, then $p_{c,n} = 1 - \frac{I_n}{n}$.

Although data independence alone does not directly translate to speedup, it provides a high-level profile of the likelihood of conflict of a set of critical regions.

4.4 Conflict density

One component of the projected speedup is the expected number of data independent critical section executions. A second component is the *density* of the conflicts. If one thinks of the conflicts as a graph, with critical regions as nodes and conflicts as edges, the density of the conflict is the number of edges per node. Intuitively, if every conflicting critical section execution conflicts with every other, one expects each conflicting critical section to execute serially, which means lower performance for optimistic synchronization. On the other hand, if one thread writes memory read by

31 other threads, all readers should complete once the writer completes, resulting in a performance gain that is commonly experienced in practice. This scenario would have a star topology if represented as a graph; removing the most connected node causes all others to become disconnected (and able to proceed concurrently). Hence, modeling this phenomenon is crucial to the accuracy of predicting performance.

Syncchar calculates the conflict density of a sample of address sets as follows. For each address set in C_n, we measure the number of conflicts with other address sets in C_n and divide it by $(|C_n|-1)$. The sum of each of these terms is the density (D_n). Formally, this is expressed as

$$D_n = \sum_{x \in C_n} \frac{\sum_{y \in \{C_n - x\}} \mathit{conflicts}(x, y)}{|C_n - x|}$$

where $\mathit{conflicts}(x, y)$ evaluates to 1 if address sets x and y can conflict and 0 otherwise. In the case where all conflicting address sets conflict with all others, $D_n == |C_n|$. In the star topology case, this should equal 2.

The expected speedup of n concurrent threads produced by this model is $(I_n) + (|C_n| - D_n)$, or the number of data independent threads plus the difference of the conflicting threads and their conflict density. According to Amdahl's law, the overall speedup obtained from optimistic execution of the critical sections associated with a given lock is constrained by the amount of time associated with that lock in the sequential execution. Thus, the projected execution time will be the time not associated with the lock plus the time associated with the lock divided by the speedup.

4.5 Limitations

Like any model, Syncchar makes certain simplifying assumptions that balance complexity against accuracy. This section lists some key cases where the Syncchar predictions can deviate from the measured performance using optimistic synchronization.

High contention and conflicts generally reduce the performance of transactional memory systems. While performance degrades under high contention, the degree of degradation depends on the implementation details of the transactional memory system. Bobba et al. [2] list a number of cases where certain hardware transactional memory design decisions can have performance pathologies under certain workloads. Because performance under high contention is largely the result of hardware features such as cache behavior and back-off strategy, which can vary substantially across implementations, Syncchar elects to model them as a simplification.

In many high-contention microbenchmarks, hardware transactional memory can achieve a substantial speedup over locking simply by eliminating cache coherence misses on the lock. In larger critical regions, however, the cost of the cache miss to acquire the lock is amortized over a long period. In larger programs, such as the Linux kernel, locks tend to be placed in the cache-aligned data structures that they protect to avoid these needless coherence misses. For instance, Ramadan et al. show a 2× speedup on the shared counter benchmark but no substantial speedup from a rote conversion of the Linux kernel to use hardware transactions [26]. Syncchar does not attempt to evaluate the quality of lock placement or its effects on optimistic performance.

4.6 Implementation details

The Syncchar prototype was developed as a module for the Simics full-system, execution-driven simulator [14] with a post-processing phase. By implementing Syncchar in the machine simulator, we are able to measure the Linux kernel as a lock-based, concurrent application *par excellence*. Nearly all instrumentation was performed using simulator breakpoints, minimally affecting the behavior of the kernel itself. The primary code changes required in the kernel itself were simulator notification via "magic instructions" when cached objects were reallocated.

The requirements of the Syncchar model are modest enough that it could be implemented for user-level applications using a binary instrumentation tool such as Pin [13], or potentially in a virtual machine monitor for kernel instrumentation on a live machine.

5. MODEL VALIDATION

To validate our model of optimistic synchronization behavior, we compared the transactional performance of three microbenchmarks to the predictions made by Syncchar. Each of the microbenchmarks has different patterns of access to shared data, described below:

- Prob - In each critical region, a single shared variable is written with 50% probability. Otherwise, work is performed on private data.
- RW - A single thread writes to a shared data structure in each critical region. All other threads read the updates in their critical regions and use the values to perform work on private data.
- RB-Tree - In each thread's critical region, a random value is inserted into or deleted from an RB-Tree. The range of values and probability of insert vs. delete are configurable. For this experiment we used a 75% probability of insertion and a maximum value of 200. The RB-Tree implementation is taken from the Linux kernel.

All benchmarks have 32 threads that execute an equal number of critical sections.

All experiments are performed using Virtutech Simics [14], modeling 8, 16, and 32 1GHz x86 processors. As Simics supports only a fixed IPC, the simulations used an IPC of 1, which is a reasonable choice for a moderate superscalar implementation. Each processor has a 16 KB, 4-way set associative private L1 cache with 64 byte lines and an access time of zero cycles. L2 caches are also private, with 64K 64-byte lines and 8-way set associativity. L2 cache accesses cost 16 cycles, and are kept coherent using a MESI snooping protocol, modeled by the gcache Simics module. Main memory is a single, shared 1 GB, with an access time of 200 cycles.

All lock-based experiments run on Linux version 2.6.16.1. We downloaded the MetaTM hardware transactional memory model and TxLinux version 2.6.16.1 for the transactional memory experiments [26, 27]. Main memory access latency is pseudo-randomly perturbed to account for performance variability, as described by Alameldeen et al. [1]. Measurements from transactional memory simulations are thus presented as a mean of 4 simulated executions.

Figure 5 shows the execution time of each microbenchmark as projected by Syncchar and the measured execution time of each benchmark using transactional memory. The predictions show a mean error of 10%, validating the effectiveness of the Syncchar model. In general, the predictions are slightly conservative, as they don't account for "lucky" interleavings, where one transaction that would have conflicted commits just before the other transaction performs the conflicting memory operation.

The model can also be more optimistic in the case of high contention. In these results, the predicted execution time for Prob at 32 CPUs was 31% worse than the measured performance. This is largely because the transactional case eliminates all cache misses on the lock variable, which dominates performance of high-contention workloads at greater CPU counts. Hardware transactional memory

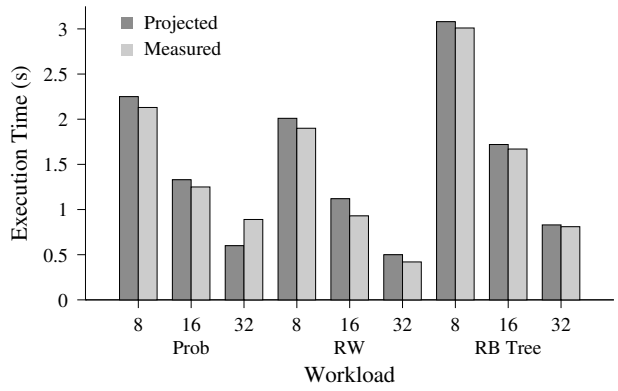

Figure 5: The projected execution time of each microbenchmark, compared with the measured execution time of the benchmark using transactions.

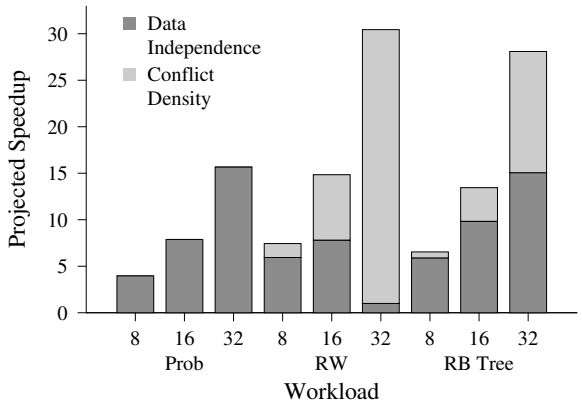

Figure 6: The projected speedups of each benchmark, decomposed into the portion attributable to data independence and conflict density.

systems can be better at backing off of a hot cache line under contention than simple spinlocks, and this reduction in cache traffic can have a substantial effect on performance that is beyond the scope of our model.

Figure 6 shows the projected speedup of each benchmark, broken down into the portion attributable to data independence and the portion attributable to conflict density. The Prob benchmark's projection, on one end of the spectrum, is entirely due to data independence. Given that all conflicting threads will be performing a write to the same location, all conflicting threads will form a serial schedule, yielding no concurrency beyond the threads that are data independent. On the other extreme, every critical section in the RW benchmark should conflict at least once, yet the measured speedup is still substantial. As discussed in Section 4.4, this is the motivating example for adding conflict density to the model. The RB-tree benchmark shows a reasonable mix of both data independence and non-dense conflicts, which we expect to be the case with more realistic workloads.

These experiments show that the Syncchar model strikes a good balance between accuracy and complexity.

```
acquire_lock();
while(!mylist.empty()){
   item = mylist.get_next_item();
   data_structure.put(item);
}
release_lock();
```

```
while(!mylist.empty()){
   item = mylist.get_next_item();
   begin_tx();
   data_structure.put(item);
   end_tx();
}
release_lock();
```

Figure 7: Pseudocode for the common idiom of holding a lock across otherwise independent work to avoid the overhead of contention, followed by an example of splitting each iteration into a separate transaction.

6. SYNCCHAR AS A TUNING TOOL

In addition to predicting the performance of an optimistic system, the Syncchar model can provide clues as to which critical sections are likely to have performance problems after a conversion. This section demonstrates the utility of Syncchar for tuning optimistic synchronization, using the (Tx)Linux kernel as case study.

6.1 Keys to tuning optimistic concurrency

Tuning the performance of optimistically synchronized code consists of increasing data independence and minimizing conflict density through careful management of address sets. In contrast, locking requires shared data to be accessed with the right locks held for correctness, but there is no performance penalty to touching *additional* shared data once a lock is held. In many cases, such as storing the size of a list, updating additional data can increase overall code efficiency. Under optimistic concurrency, however, minimizing access to shared data is a first-order performance concern.

Per-CPU data structures are the kernel analog to per-thread data structures in user-level programming. Memory allocators, for instance, typically reserve per-CPU caches of memory to avoid contention for the common pool of free memory. Similarly, if a data structure contains an integer field that is mostly written, having a per-CPU copy can eliminate a conflicting element from the address sets of many transactions.

The downside to these approaches is that they introduce extra work and conflict exposure for aggregation or redistribution. If one can eliminate conflicts on shared fields and increase data independence, however, the extra work in the single threaded case can be offset by additional concurrency.

Reorganizing data structures that have a single access point (e.g. a linked-list) into ones that uniformly distribute the data over a number of entry points (e.g. a hash table) is another method for avoiding data conflicts. By replacing a list's single head pointer with an array of hash buckets, the probability that an insertion will conflict is reduced to 1 over the number of buckets. Not all linked-lists can be replaced with hash tables, as some rely on the specific semantics of a list, such as ordering. In these cases, other data structures that provide multiple access paths to the data should be explored.

Another optimization technique for optimistic concurrency is splitting longer critical sections into shorter ones. Consider a function

that does a bulk insertion of items into a data structure, as illustrated in Figure 7. After inserting any given item, all data structure invariants hold. In a locking environment, holding the lock across all iterations is generally most efficient because it avoids bus locking and cache traffic on the lock variable itself (potentially at the cost of fairness). Under optimistic synchronization, however, one should avoid touching larger amounts of data for longer periods of time and instead should commit after each iteration. This is depicted in the second half of Figure 7 by the reduced scope of the transaction relative to the scope of the lock. In addition to limiting exposure to conflicts, reducing the live range of a transaction also limits the amount of work lost in the event of a rollback.

In cases where critical regions have dense conflicts that cannot be optimized, the best course of action may be to fall back on locking. In some cases, a highly contended shared pointer or variable is integral to the algorithm and cannot be changed. Ideally, optimistic synchronization would do no worse than locking when critical regions have dense conflicts. In transactional memory systems, however, retrying a transaction incurs the overhead costs including cache interference and back-off periods that can make overhead of acquiring a lock cheaper under high contention.

6.2 Where to start tuning Linux?

The Linux kernel, like other real-world parallel programs, is a large and complicated piece of software with hundreds of critical sections that could be tuned after a rote conversion to use optimistic concurrency. Targeting effort on the critical sections that are most likely to benefit from tuning is a major challenge.

In particular, the best opportunity is for coarser-grained locks that represent a substantial portion of execution time and protect data structures that can be easily reorganized to avoid conflicts. The problem with further tuning fine-grained locks is that they will have low data independence and there is very little that optimistic concurrency has to offer such a situation.

We profiled the kernel using the Syncchar tool, with Table 2 showing the data independence and conflict density of some of the longest held locks during the MAB workload on 16 processors. This sample is representative of all of our workloads; relative positions and length held vary but these locks are generally in the top 15 on all workloads. The locks that tend to be held the longest tend to be coarse-grained, global locks, so it shouldn't come as a surprise that these dominate the table.

Overall, the Linux kernel has fairly low data independence—less than 2%, weighted by the time each lock is held. The conflict density is more moderate, accounting for just over a fourth of potential conflicts. This indicates that there is a good deal of opportunity for tuning the optimistic performance of the kernel, by changing data structures to increase data independence and changing the scope of critical regions to reduce conflict density. The absolute performance increases will be small as all spinlocks only account for 2% of the runtime in this workload.

We can gain further insight into a lock's potential for optimization by looking at the distribution of conflicts within Syncchar's sampled worksets. Table 3 shows the distribution of conflicting addresses for the zone allocator locks listed in Table 2. For instance, the `zone.lru_lock` shows a single conflict on a hot-spot address, and a large number of unique addresses that are rarely involved in conflicts. On the other hand, if a larger portion of the address set is involved in most of the conflicts, or if a highly conflicting address is critical to maintaining data structure invariants, the critical section may not be amenable to further tuning, or may perform better under conservative synchronization.

Lock	Time Held	Data Indep. 16	Data Indep. 32	Confl. Dens. 16	Confl. Dens. 32
dcache_lock	0.34%	8.12	12.54	2.30	3.43
zone.lru_lock	0.31%	1.15	1.23	15.04	29.48
inode.i_data- .i_mmap_lock	0.26%	4.87	6.85	6.09	10.24
files_lock	0.20%	5.76	9.21	7.42	13.06
zone.lock	0.08%	1.00	1.00	16.00	32.00
kernel_flag	0.08%	7.98	13.06	3.75	6.68
journal_t.j_state_lock	0.07%	1.01	1.04	15.82	31.46
inode_lock	0.07%	3.02	5.47	9.83	18.58
Weighted Average	2.00%	1.38	1.93	4.11	7.45

Table 2: A sample of the longest held spin locks during the MAB workload on a 16 CPU machine. Each entry is for one instance of a lock. The percentages of time held are of total execution time. Data independence and conflict density measurements are given for each lock at 16 and 32 CPUs. The final row gives the total percent of the execution time spent holding any spin lock, and then provides an average data independence and conflict density measurement for the kernel, weighted by the length of time each lock is held.

Lock	zone.lru_lock	zone.lock
0-19%	4,012	2,188
20-39%	1	3
40-59%	0	0
60-79%	0	0
80-99%	1	1
100%	0	2
Avg. Workset Size	4,012	2,188
Total Conflicts Sampled	225,482	28,938

Table 3: Distribution of conflicting addresses for the zone locks. The range of percentages shows the number of workset addresses involved in that percentage of conflicts.

6.3 Zone allocator case study

We selected the Zone allocator for our case study because its critical sections are highly likely to conflict, its locks are among the longest held in the kernel, and its conflict profile indicates only a few "hot" memory locations out of an otherwise well-distributed address set. Linux divides physical memory into zones (3 on the x86 architecture), and uses these `zone` structures to manage the allocation and freeing of physical memory.

There are two locks in each `zone` structure that protect different structures. The `zone.lock` protects the lists of free pages, which are used to implement a buddy allocation algorithm. The `zone.lru_lock` protects the least-recently-used (LRU) lists, which are used to select pages that can be reclaimed or swapped out on demand. Both lists have a similar conflict profiles—shared list pointers and size counters.

The data structures protected by the `zone.lru_lock` are amenable to reorganization to avoid conflict. The first issue with the LRU data is that it is stored in linked-lists. To solve this problem, we converted the single lists to hash tables with chaining, as described in Section 6.1. The page frame reclaiming algorithm (PFRA) does require a rough ordering for performance, but neither performance nor overall effectiveness are compromised if items aren't exactly ordered. The original PFRA scans the free lists starting at the tail

configure	Run several parallel instances of the configure script for a large software package, one for each processor.
find	Run 32 instances of the `find` command, each in a different directory, searching files from the Linux 2.6.16 kernel for a text string that is not found. Each directory is 4.6–5.0MB and contains 333–751 files and 144–254 directories.
MAB	File system benchmark simulating a software development workload [20]. Runs one instance per processor of the Modified Andrew Benchmark, without the compile phase.
pmake	Runs make -j 2 * number_of_procs to compile 27 source files totaling 6,031 lines of code from the libFLAC 1.1.2 source tree in parallel.

Table 4: Parallel applications used to exercise the concurrency in Linux and TxLinux.

pointer, which we replace with the index of the last hash bucket scanned. This index is updated with atomic instructions outside of the critical regions to avoid conflicts and allow parallel scanning of the list.

The `zone.lru_lock` also protects integers that track statistics such as the number of pages scanned. Some of these counters are accessed both inside and outside of critical regions, implying no strict consistency requirement. These updates are moved outside of the critical regions. The integers that had consistency requirements, however, were converted into per-CPU counters.

The `zone.lock`, however, is less amenable to reorganization because the buddy allocator algorithm is concerned with both performance and avoiding memory fragmentation—these concerns can work at cross purposes in optimistic systems. Early results indicate that partitioning the free lists into per-CPU lists is not as effective as one might expect. In order to maintain the same aggressive level of fragmentation avoidance, a CPU often has to search another CPU's free list for a block of the right size before splitting a larger one. Changing this behavior requires evaluation not only of execution time but also of the change in memory fragmentation. Developing a higher-performance optimistic zone allocator is ongoing work.

One opportunity we found in the the code protected by the `zone.lock` is decomposing each iteration of the bulk allocation and free operations into smaller critical regions, as described in Section 6.1. Because the `zone.lock` shows such poor data independence and conflict density, we also optimized it by converting its critical regions from transactions back into spinlocks.

6.4 Experimental Results

To evaluate these optimizations, measurements of both 16 and 32 CPU systems were taken as described in Section 5. Because Linux simply idles when left undisturbed, we need a set of parallel applications to exercise the concurrency within the Linux subsystems, such as the file system and memory allocator. The applications used for this study are listed in Table 4.

Figures 8 and 9 present the speedup in kernel time at 16 and 32 CPUs, respectively. Kernel time is presented because our tuning efforts are for the kernel qua parallel program and cannot be expected to improve the non-kernel portions of the execution time. Further, this approach eliminates noise introduced by load imbalance in the workloads.

These graphs compare the optimizations against the TxLinux baseline kernel. "zone coarse tx" represents the LRU optimizations and the `zone` transactions unmodified. "zone fine tx" includes both the LRU optimizations and the shortened `zone` bulk allocations and free transactions. "zone coarse lock" reverts the `zone.lock` back to a spinlock and does not shorten the bulk critical regions. Table 5 summarizes the optimizations described above and their mapping onto labels in the figures.

At 16 CPUs, the optimized kernels perform worse than the unoptimized in all but the find benchmark, whereas at 32 CPUs, they do better than the baseline (except config). Only config performs worse than the baseline, but within 1% of the baseline. Our modifications add work to each critical section in order to avoid conflicts. Thus, each critical section is longer, but more of the work can be done in parallel, ameliorating this cost at higher CPU counts.

TxLinux	Default TxLinux kernel
zone coarse tx	The `lru_lock` optimizations and the `zone.lock` transactions unmodified
zone fine tx	The `lru_lock` optimizations and the bulk `zone.lock` transactions shortened
zone coarse lock	The `lru_lock` optimizations, while the `zone.lock` transactions are reverted back to locks. The bulk `zone` critical sections are not shortened.

Table 5: Summary of `zone` optimizations evaluated.

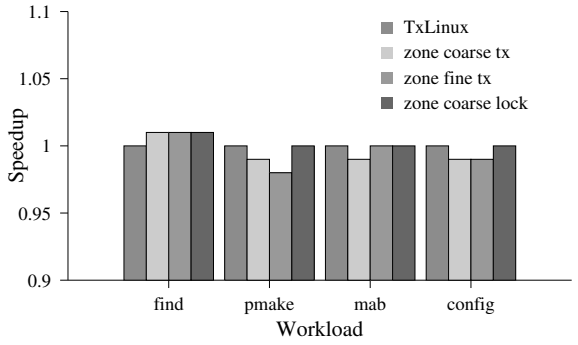

Figure 8: Speedup of time spent in the kernel for each benchmark at 16 CPUs, compared to the TxLinux baseline (larger is better).

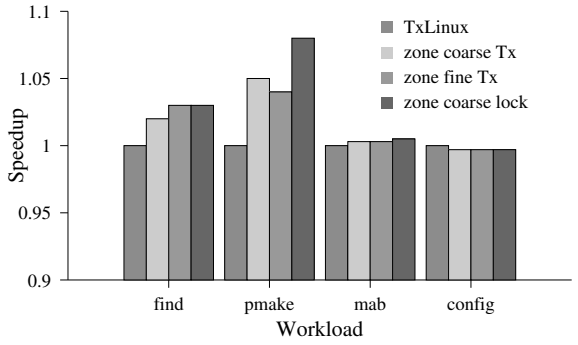

Figure 9: Speedup of time spent in the kernel for each benchmark at 32 CPUs, compared to the TxLinux baseline (larger is better).

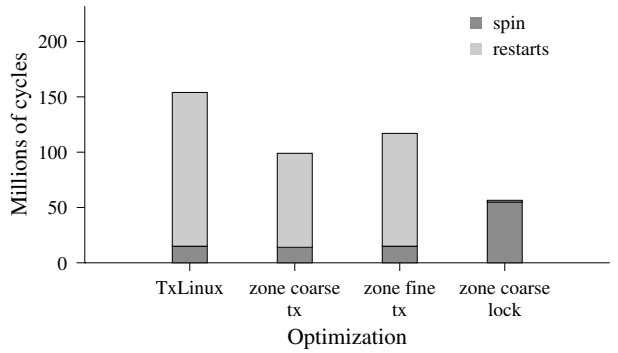

Figure 10: Breakdown of synchronization time between spinning on a lock and restarting transactions for each kernel optimization, running pmake at 32 CPUs.

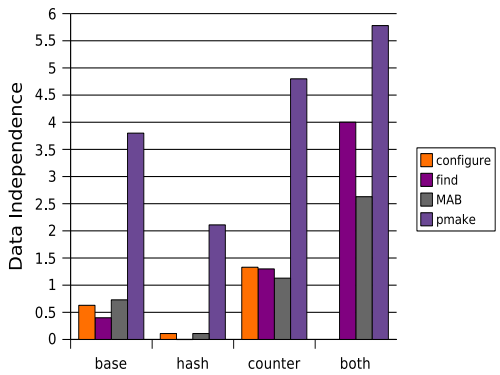

Figure 11: Change in data independence for each LRU optimization at 32 CPUs (larger is better). Base is TxLinux. Both is the combination of the changes. Configure (combined) data were unavailable at time of submission.

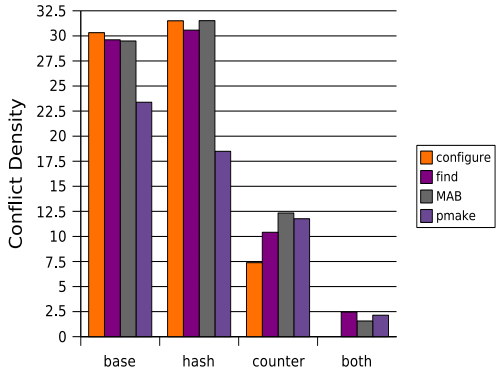

Figure 12: Change in conflict density for each LRU optimization (smaller is better) at 32 CPUs. Base is TxLinux. Both is the combination of the changes. Configure (combined) data were unavailable at time of submission.

Figure 10 shows the distribution of synchronization time across each kernel optimization. It turns out that the `zone` transactions dominate most of the transaction restarts, and that this pattern closely matches the execution time of pmake. Unlike the other benchmarks, pmake generates enough contention for the data structures protected by the `zone.lock` that the "zone fine tx" optimization performs worse than the others. This is largely the result of backing off many times in an exponential back-off scheme. While this optimization reduces the likelihood of conflict, it does not eliminate it. Under higher contention workloads, these long back-off periods can waste a lot of time. can waste a lot of time due to long back-offs.

The "zone coarse lock" optimization reverts the `zone` transactions back to using spinlocks for isolation. This optimization eliminates nearly all of the aborted cycles in the TxLinux kernel and performs the best under highest contention. This sets the standard for an optimistically synchronized, data independent zone allocator to beat.

Figure 11 shows the data independence for the zone LRU list optimizations, and Figure 12 shows the change in conflict density. Making counters per-CPU is a clear win, but adding the hash table replacement alone actually lowers the data independence and raises conflict density because it adds more work to many code paths without removing the conflicts on the counters. When combined with the counter optimization, however, the hash table increases data independence and lowers conflict density more than the counter optimization alone.

The data independence and conflict density data for these LRU optimizations indicate that the likelihood that a critical section will restart more than once is lowered much more than the likelihood that it will not restart at all. While data independence is improved in these optimizations, the more dramatic improvement comes from a reduction in conflict density.

Data independence and conflict density measurements for the zone optimizations are not listed, as they were not restructured in a way that would affect these quantities.

This case study indicates that tuning the performance of optimistic synchronization can be tricky and nonintuitive; programmers facing this challenge will benefit immensely from tools that help them understand their system with quantitative rigor.

6.5 Application to the Linux Kernel

In addition to tuning the performance of TxLinux, we found that decomposing the bulk operations of the `zone.lock` into shorter critical sections improves the performance of Linux as well. Our intuition is that this patch prevents smaller requests from waiting on larger ones. While grabbing and releasing the lock within the loop adds a few instructions, it can lower the latency for a particular thread's allocation which is often on the workload's critical path. Lowering the average latency for allocation can increase system throughput. This change has been submitted to the Linux kernel developers, who are interested in incorporating the change with the upcoming addition of ticket spinlocks.

We verified these results by applying this change to the 2.6.23.1 kernel and ran various parallel workloads on a Dell PowerEdge 2900 with two Quad-core Xeon chips, each core operating at 2.66 GHz. The test system has 8 GB of RAM. The test benchmarks are listed in Table 6.

Figure 13 shows the performance for each benchmark on the target system, normalized to the unmodified performance. In all cases except for the larger compilation (with only an .8% loss), the optimized version outperformed the baseline Linux kernel. The best performance was on the shorter compile, with a 4.8% improvement.

In this example, Syncchar shows its versatility by finding opportunities for performance improvements both in optimistically synchronized code as well as conservatively synchronized code.

configure	Run several parallel instances of the configure script for a large software package, one for each processor.
hdparm	Execute `hdparm -t /dev/sda1` to check for regressions on the IO processing path. Unlike the others, which report seconds, this reports MB/s.
Kernel Compile	Execute `make -j 16` on the 2.6.23.1 kernel source with default Kconfig options.
Kernel Compile (fast)	Execute `make -j 16` on the TxLinux kernel source, which has many fewer Kconfig options selected.
MAB	File system benchmark simulating a software development workload. [20] Runs one instance per processor of the Modified Andrew Benchmark, without the compile phase.
objdump	Execute `objdump -d -l vmlinux | grep mov >/dev/null` on the 2.6.23.1 kernel image.

Table 6: Parallel applications used to evaluate the Linux kernel patch.

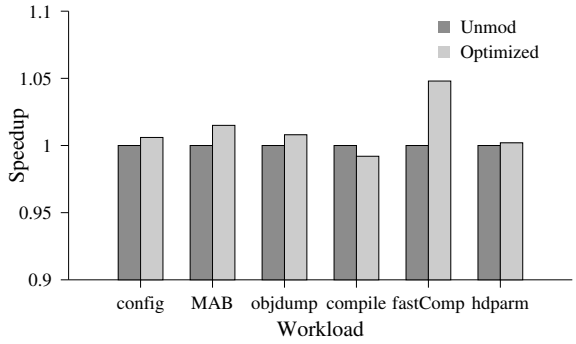

Figure 13: Speedup for each benchmark on both the unmodified Linux kernel (unmod) and the kernel with the split zone bulk operations (Optimized).

7. RELATED WORK

This paper employs techniques from parallel programming tools to evaluate the limits of optimistic concurrency. There is a large body of previous work on debugging and performance tuning tools for lock-based programs [5, 30]. Our work is distinguished from other tools because it augments these techniques with novel methods for reasoning about performance issues in optimistic systems.

Porter et al. introduce the definition of data independent critical sections and a tool for measuring data independence of critical regions, but provide no concrete application of the metric or experimental evaluation of its utility [22]. This paper substantially revises the data independence metric to be more practical (both in measurement and utility), introduces the new metric of conflict density, and provides empirical evaluation of its application to prediction and tuning.

When memory performance became a big issue in the early 90's, programmers needed tools like *memspy* [15] to reason about performance problems related to memory behavior. Now that parallel architectures are unavoidable, programmers will need tools to assist in tuning the performance of optimistically synchronized systems. Lev and Moir discuss the necessity for and challenges of debugging tools for transactional memory systems [12]. This paper is distinguished by addressing performance, whereas their work addresses correctness problems. Perfumo et al. introduce a Haskell runtime with transactional memory instrumentation support for performance profiling [21]. This work is complementary to the Syncchar model, which provides limits that are not tied to specific schedules.

7.1 Transactional memory

Transactional memory is a general form of optimistic concurrency that allows arbitrary code to be executed atomically. Herlihy and Moss [9] introduced one of the earliest transactional memory systems. More recently, Speculative Lock Elision [23] and Transactional Lock Removal [24] optimistically execute lock regions transactionally. Several designs for fully-functional transactional memory systems have been proposed [4, 17, 25]. Larus and Rajwar provide a more complete survey of transactional memory literature [11].

Welc et al. show that the Java monitor abstraction can be adaptively changed between mutual exclusion and transactions [29]. Their work is limited to this programming model and, while potentially applicable to others, is not a general purpose solution to the problem of deciding whether to convert an application to use transactions.

7.2 Alternative forms of optimism

Lock-free (and modern variants like obstruction-free) data structures are data-structure specific approaches to optimistic concurrency [6, 8]. Lock-free data structures attempt to change a data structure optimistically, dynamically detecting and recovering from conflicting accesses. Lock-free data structures, while optimistic, are not a general purpose solution. Lock-free data structures require that each data structure's implementation meets certain nontrivial correctness conditions. There is also no general method to atomically move data among different lock-free data structures.

The Linux kernel employs a form of optimistic concurrency with the *seqlock*, or sequence lock [3]. Seqlocks allow readers to execute optimistically by reading a sequence number before and after reading the protected data. Writers similarly write the sequence number before and after writing a data structure, and writers lock each other out. If the sequence number has the same, even value before and after reading, the readers are assured the data they read was consistent. If the value changes, or is odd (indicating the presence of a writer) the readers simply retry until they read a consistent value.

The Linux kernel's Read-Copy Update technique [16] is also a variant on lock-free data structures that uses compare-and-swap instructions to atomically replace pointers to complex data structures. Old copies of data structures are cleaned up by the process scheduler after all reader processes have left the kernel.

8. CONCLUSION

This paper introduces a novel method and tool for reasoning about the performance of optimistic synchronization, based on measurements from lock-based code. We have validated its effectiveness at performance prediction and demonstrated its usefulness in guiding performance tuning on both the Linux and TxLinux kernels. We see Syncchar as one in an array of profiling and debugging tools that will help application developers leverage multicore systems more effectively.

9. REFERENCES

[1] A. Alameldeen and D. Wood. Variability in architectural simulations of multi-threaded workloads. In *HPCA*, 2003.

[2] J. Bobba, K. E. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift, and D. A. Wood. Performance pathologies in hardware transactional memory. *SIGARCH Comput. Archit. News*, 35(2):81–91, 2007.

[3] D. Bovet and M. Cesati. *Understanding the Linux Kernel*. OŔeilly Media, Inc., 3rd edition, 2005.

[4] L. Hammond, V. Wong, M. Chen, B. Carlstrom, J. Davis, B. Hertzberg, M. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *ISCA*, 2004.

[5] J. Harrow. Runtime checking of multithreaded applications with visual threads. In *SPIN*, pages 331–342, 2000.

[6] M. Herlihy. Wait-free synchronization. In *TOPLAS*, January 1991.

[7] M. Herlihy and E. Koskinen. Transactional boosting: A methodology for highly-concurrent transactional objects. *Technical Report: CS07-08, Brown University Computer Science*, pages 1–17, July 2007.

[8] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *ICDCS*, 2003.

[9] M. Herlihy and J. E. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, May 1993.

[10] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 211–222, New York, NY, USA, 2007. ACM Press.

[11] J. R. Larus and R. Rajwar. *Transactional Memory*. Morgan & Claypool, 2006.

[12] Y. Lev and M. Moir. Debugging with transactional memory. In *TRANSACT*, 2006.

[13] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[14] P. Magnusson, M. Christianson, and J. E. et al. Simics: A full system simulation platform. In *IEEE Computer vol.35 no.2*, Feb 2002.

[15] M. Martonosi, A. Gupta, and T. A. Anderson. Memspy: Analyzing memory system bottlenecks in programs. In *Measurement and Modeling of Computer Systems*, pages 1–12, 1992.

[16] P. E. McKenney. *Exploiting Deferred Destruction: An Analysis of Read-Copy Update Techniques in Operating System Kernels*. PhD thesis, 2004.

[17] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based transactional memory. In *HPCA*, 2006.

[18] M. J. Moravan, J. Bobba, K. E. Moore, L. Yen, M. D. Hill, B. Liblit, M. M. Swift, and D. A. Wood. Supporting nested transactional memory in logtm. In *ASPLOS-XII*. 2006.

[19] J. E. B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. PhD thesis, 1981.

[20] J. K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *USENIX Summer*, 1990.

[21] C. Perfumo, N. Sonmez, A. Cristal, O. Unsal, M. Valero, and T. Harris. Dissecting transactional executions in haskell. In *TRANSACT*, 2007.

[22] D. Porter, O. Hofmann, and E. Witchel. Is the optimism in optimistic concurrency warranted? In *HotOS*, 2007.

[23] R. Rajwar and J. Goodman. Speculative Lock Elision: Enabling highly concurrent multithreaded execution. In *MICRO*, 2001.

[24] R. Rajwar and J. Goodman. Transactional lock-free execution of lock-based programs. In *ASPLOS*, October 2002.

[25] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *ISCA*. 2005.

[26] H. Ramadan, C. Rossbach, D. Porter, O. Hofmann, A. Bhandari, and E. Witchel. MetaTM/TxLinux: Transactional memory for an operating system. In *ISCA*, 2007.

[27] C. Rossbach, O. Hofmann, D. Porter, H. Ramadan, A. Bhandari, and E. Witchel. Using and managing transactional memory in an operating system. In *SOSP*, 2007.

[28] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A scalable approach to thread-level speculation. In *ISCA*, 2000.

[29] A. Welc, A. L. Hosking, and S. Jagannathan. Transparently reconciling transactions with locking for java synchronization. In *ECOOP*, 2006.

[30] Y. Yu, T. Rodeheffer, and W. Chen. Racetrack: Efficient detection of data race conditions via adaptive tracking. In *SOSP*, 2005.

[31] C. Zilles and L. Baugh. Extending hardware transactional memory to support non-busy waiting and non-transactional actions. In *ACM SIGPLAN Workshop on Languages, Compilers, and and Hardware Support for Transactional Computing*, Jun 2006.